Journal of Interconnection Networks

SUPPORTING RUNTIME RECONFIGURATION ON NETWORK PROCESSORS

KEVIN LEE and GEOFFREY COULSON
Computing Department, InfoLab21, Lancaster University,
Lancaster, LA1 4WA, UK
{*leek,geoff*}*@comp.lancs.ac.uk*

Network Processors (NPs) are set to play a key role in the next generation of networking technology. They have the performance of ASIC-based routers whilst offering a high degree of programmability. However, the programmability potential of NPs can only be realised with appropriate software. In this paper we argue that specialised software to support *runtime reconfiguration* is needed to fully exploit the potential of NPs. We first justify supporting runtime reconfiguration on NPs by offering real-world scenarios and discussing the issues associated with these. We then demonstrate how runtime reconfiguration can be achieved in practice through a case study of our component-based programming approach on the Intel IXP2400 NP.

Keywords: Network Processor; Component; Runtime Reconfiguration; Programming Model.

1. Introduction

Network Processors (NPs) are multiprocessor-based hardware units that have the ability to perform relatively complex packet processing tasks in software at line speeds when compared to contemporary devices. They typically consist of a set of heterogeneous processors including packet processors, dedicated devices such as hashing or encryption engines, general purpose processors, and a high-speed interconnect [3]. These can be supported either on a single chip or across multiple chips.

NPs can be seen as an attempt by hardware vendors to fulfill the growing need for network hardware elements that support high throughput while also offering increased programmability. Programmability is seen as crucial in supporting system evolution so that new protocols and services can be accommodated without designing new hardware. In addition, their programmability makes NPs very widely applicable—e.g. they are being applied in networked devices, as edge-network routers and even in the network core [13].

In addition, it is now becoming recognised [20] that *runtime reconfiguration* is a desirable characteristic of software for NPs. Runtime reconfiguration is useful for a number of applications, including dynamically extensible services [16], network resource management [10], configurable network-based encryption [12], and offloading of processing [9]. In addition the active networking (AN) community have been heavily

involved in investigating the use of NPs in their field [14]. This is because ANs require significant data-plane processing and also require routers to expose their state to allow reconfiguration of forwarding functions.

The aim of the research discussed in this paper is to investigate the potential and benefits of runtime reconfiguration in NPs. Our research focuses on the provision of generic mechanisms that can potentially be applied in a wide range of scenarios including all of the above. Essentially, we adopt a runtime component-based approach in which fine-grained components on the NP can be dynamically (un)loaded and (dis)connected in a principled manner. In this paper we illustrate the generality of our approach by focusing on applying it in a set of representative scenarios. We use the Intel IXP 2400 [6] as representative of the state of the art in NPs. We also argue that a flexible runtime reconfiguration capability need not be bought at the expense of performance.

The remainder of this paper is structured as follows. In section 2 we motivate runtime reconfiguration by outlining a few reconfiguration scenarios. Section 3 then provides background on the Intel IXP2400 and its current lack of software support for runtime reconfigurability. Section 4 then presents details of OpenCOM, our runtime reconfiguration capable programming platform, and its deployment on the IXP2400. In section 5 we show how the scenarios of section 2 can be realised by OpenCOM. Finally, we discuss related work in section 6 and in section 7 offer our conclusions and discuss future work.

2. Runtime Reconfiguration

In this section, we present a number of real-world runtime reconfiguration scenarios. Some of these introduce new services at runtime which in the past would have required the system to have been taken off-line and would perhaps have required additional hardware. Others introduce fine grained adaptation mechanisms that would not have been possible without custom hardware. We refer again to the first two of these scenarios in section 5.1 when we illustrate their realisation using our component-based programming model.

2.1. *Dynamic Proxying*

In general terms, proxying is a technique for allowing clients and devices to make indirect connections to network services via a shared intermediary. It is used both to limit the network load incurred in providing access to external network resources and to provide value-added services. The proxy notion can be applied in a wide range of settings including web caching, VoIP proxying, and media transcoding. Furthermore, it can involve a range of generic techniques including combining client requests, diverting connections, denying connections, or creating encrypted tunnels.

Currently, proxying is typically performed at the network edge on dedicated devices. However, with NPs it becomes possible to deploy proxies on routers inside the

network. Furthermore, such proxies could be deployed on the fly and on demand. To support such dynamic proxys, a NP would need a software framework that incorporated a extensible classifier to identify specific flows, plus the ability to instantiate proxy components depending on the service required. The benefits would be a minimisation of latency as well as a maximisation of flexibility. Deployment overhead could also be minimised by caching proxies on the NP.

2.2. *Adaptive Load Balancing*

On standard network routers, flows are either not differentiated or are differentiated in a relatively static manner (e.g. using diffserv [1]). There is no capability to adapt the resources dedicated to different flows in a fine grained manner depending on current application needs or traffic patterns.

With NPs, however, it is possible to dynamically deploy resources to different flows. For example, a given number of hardware threads, or packet processors, could be dedicated to high-priority VoIP flows, depending on patterns of demand (e.g. as a function of the time of day). As well as packet forwarding, this also applies to per-flow processing such as in-band transcoding. Furthermore, because NPs have the ability to process and forward traffic while simultaneously analysing the traffic to determine a suitable load balancing policy, they offer the ability to perform "intelligent" load balancing. In contrast to off-line (policy based) load balancing, fine-grained load-balancing mechanisms can range from diverting flows to different routes to replicating processing/classification code across multiple packet processors.

2.3. *Additional Scenarios*

We list here a few more noteworthy reconfiguration scenarios.

i) Fast mobile handoff

Todays wireless networks have highly segmented network architectures. A consequence of this is that handoffs become more frequent as users move between cells, which in turn implies that faster and smoother network handoffs are become increasingly important to shield users from service disruption [17].

One approach to optimising handoffs in this way is to provide support on NPs within the network infrastructure [17]. On receiving a handoff request from a mobile device, the NP instantiates a short-lived "flow routing component" to temporarily divert flows directed to the mobile device's old location to its new location until the IP-level handoff has completed.

ii) IPv6 translator

Users wishing to upgrade from IPv4 to IPv6 are faced with the problem of migrating to the new protocol whilst still supporting the old. The usual way of proceeding is to either deploy the new protocol in parallel or to introduce dual protocol routers. An alternative is to migrate networks incrementally to the new protocol and use routers with IPv4-IPv6 protocol translation functionality [11].

Rather that purchase new hardware, an NP-based solution would allow IPv4-IPv6 translators to be dynamically deployed on new IPv6 subnets. An NP currently deployed as a IPv4 router would, on detection of IPv6 traffic instantiate a IPv6-IPv4 translator. All IPv6 traffic would then be directed to the translator, which would translate it to IPv6 and forward it to the IPv4 network. The translator could be removed when the IPv4 traffic has ceased. Dynamic deployment of such functionality would allow the router to operate at full speed whilst performing its primary protocol forwarding.

3. Background on the Intel IXP2400

3.1. *The Intel IXP2400*

The Intel IXP2400 NP [6] consists of a single embedded RISC processor (an Intel XScale), and eight packet processors called "micro-engines". Its architecture can be considered typical of the current generation of NPs.

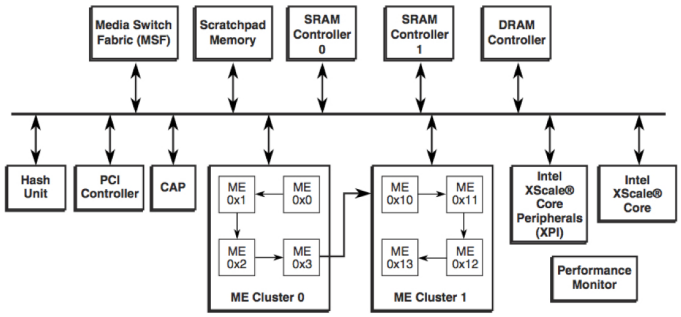

Fig. 1. The Intel IXP2400 (from [9])

The IXP2400 provides a fast bus for communication between its microengines, MAC ports and memory. It also provides shared registers and a range of memory types (i.e. SRAM, SDRAM). In addition, it provides 'next-neighbour' registers that provide a dedicated interconnect between two 'adjacent' microengines.

The microengines themselves are 233-600MHz CPUs whose instruction set provides for I/O to/from MAC-ports, packet queuing support, and checksumming. They support hardware threads with zero context switch overhead and can be programmed either in assembler or C.

In normal operation, the IXP2400 uses the microengines to support the data plane and the more general XScale to support the control plane. The shared registers and memory are typically used together at the software level to realise inter-processor communication.

Figure 1 illustrates the main building blocks of the NP including various memory types, hardware assists, control units, the XScale CPU and the microengines.

3.2. *Software for the IXP2400*

Intel's *MicroACE* [4] is an NP-based programming platform targeted at the IXP2400 and other Intel IXA products. The MicroACE model is that proxy-like software elements (called *active computing elements* or ACEs) on the IXP2400's XScale control processor are 'mirrored' by blocks of code (called microblocks) that run on microengines. Thanks to this mirroring, when the programmer loads a XScale element, the corresponding microblock is transparently loaded onto a microengine as a side effect. The microblock can choose to offload packets to its associated ACE for handling on the slow path.

Although it provides a useful degree of abstraction, the MicroACE model is *static* in nature. It does not support any type of runtime reconfiguration. Furthermore, linkages between microblocks are implicit in the way the microblocks are written. Thus, it is not possible to combine microblocks in unanticipated topologies or to exploit interconnect mechanisms other than those explicitly chosen by the microblock author. Also, the ACE approach takes no account of the *integrity* of a running configuration: if a component were to be replaced in some ad-hoc manner, a neighbouring component will inevitably fail as components expect to interact directly.

Apart from MicroACE, there is additional proprietary and non-proprietary commercial software for the IXP2400 (e.g. [58]). None of this, however, has any support for runtime reconfiguration.

4. OpenCOM

4.1. *Programming Model*

OpenCOM [7] is a language independent component-based programming platform for building low-level systems software.

A high-level view of the OpenCOM programming model is given in figure 2. This depicts the central concepts of *components* (the filled circles), *capsules* (the outer dotted box), *caplets* (the inner dotted boxes), *interfaces* (the small circles), *receptacles* (the small cups), and *bindings* (the implied association between the adjacent interfaces and receptacles).

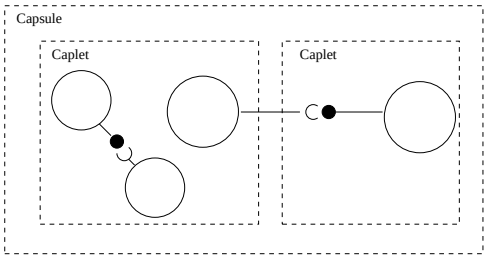

Fig. 2. Illustration of capsules and caplets

- **Components, capsules and caplets** *Components* are encapsulated units of

functionality and deployment that interact with their environment (i.e. other components) exclusively through interfaces and receptacles. Components carry negligible inherent overhead and can effectively be used in extremely fine grained compositions. Crucially, OpenCOM is a *runtime* component model meaning that (unlike, say, NP-Click [18]) components can be dynamically deployed at any time during run-time. The locus of component deployment is either a *capsule* or a *caplet*; the latter are subscopes of the former. Different caplets can also host components written in different *component styles*. Component styles are different system-level implementations of components which may have different representations and different semantics (e.g. because they run on different CPU types). Nevertheless, all styles still look the same to external third-party code that loads and binds components in the standard manner supported by OpenCOM. Accommodating heterogeneous component styles enables OpenCOM to transparently support multiple deployment environments in the same capsule environment. For example, it supports co-existence between C++ and Java components or, more to the point, it supports a seamless environment that includes both the XScale processor and microengines on the IXP2400.

Each capsule offers a simple run-time API for component life-cycle management (i.e. loading components into the capsule and instantiating and destroying them), and interface/ receptacle binding (see below). To accomplish loading, the model supports the notion of *plug-in loaders*. New loaders with different behaviours can be added at runtime, and they can be selected according to their particular properties. Examples are given below. The loading of components into a capsule can be requested by any component hosted by the capsule no matter which caplet it resides in (this is referred to as *third-party deployment*).

• **Interfaces and receptacles** *Interfaces* are units of service provision offered by components; they are expressed in terms of sets of operation signatures and associated data-types. For language independence, OMG IDL is used as a specification language (note that this does *not* imply an overhead of CORBA-like stubs and skeletons!). Components can support multiple interfaces: this is useful in recognising separations of concerns (e.g. between base functionality and management). *Receptacles* are 'anti-interfaces' used to make explicit the dependencies of components on other components. Receptacles are key to supporting a third-party style of composition (to complement the third-party deployment referred to above): when third-party-deploying a component into a capsule, one knows by looking at the component's receptacles precisely which other component types must be present to satisfy its dependencies.

• **Bindings** Finally, *bindings* are associations between a single interface and a single receptacle that reside in a common capsule (but not necessarily a common caplet). Similarly to plug-in loaders, OpenCOM also supports a notion of *plug-in binders*. The idea is to give access to a range of binding mechanisms with varying characteristics. See below for examples. As mentioned, the creation of bindings is inherently third-party in nature; it can be performed by any party within the

capsule (i.e. not only by the 'first-party' components whose interface or receptacle participates in the binding).

4.2. *Higher-level abstractions*

Above the granularity of individual components, a key pattern employed in OpenCOM programming is to construct applications or systems in terms of *component frameworks* (CFs). CFs are tightly-coupled sets of components that work together to address some specific area of functionality. They accept 'plug-in' components, deployed at runtime, which somehow modify the CF's behaviour. CFs also impose constraints on their plug-ins to guard against nonsensical compositions. As an example, consider a "protocol stack" CF which accepts protocol components as plug-ins, and constrains these plug-ins to be composed linearly. In addition, it is the CFs responsibilty to maintain and transfer state consistently when reconfigurations occur. A CF that we employ specifically in NP environments, the Router CF, is discussed in section 4.5.

We also support a number of generic services that facilitate the construction of complex systems. These are themselves implemented in terms of components and are thus optional in any given capsule configuration. Key among these is a set of *reflective meta-models* [7] that facilitate dynamic reconfiguration of systems by permitting different system aspects to be inspected, adapted and extended at runtime. As examples: the *architecture meta-model* exposes the compositional topology of the components in a capsule in terms of a causally-connected graph structure; the *interface meta-model* allows one to discover information about interface types at runtime and to invoke interface instances that are dynamically discovered at runtime; and the *interception meta-model* allows one to interpose interceptors at bindings between component interfaces.

4.3. *OpenCOM on the Intel IXP2400*

We now consider how the above-described OpenCOM concepts are applied in NPs such as the IXP2400. First, the scoping-related notions of capsules and caplets are useful in distinguishing different processors and types of processors on the NP in a generic manner. Thus we map a single capsule onto the entire NP, and sub-scope individual microengines, and the XScale control processor, as caplets. The capsule runtime reside on the XScale where it runs in a standard operating system environment. Microengine caplets are implemented on the bare microengine hardware. An alternative mapping could encapsulate all the microengines within a single caplet. Then, a plug-in loader associated with this caplet could perform intelligent load balancing of components across microengines, thus providing a higher level of abstraction than the first alternative. The notion of caplets is also useful in isolating untrusted code, which is important in active networking environments. For example, a Java sandbox could be isolated as a caplet on the XScale.

The pluggable loader concept is closely associated with capsules/caplets. Typically, at least one loader is provided for each type of caplet, and each will know how to load components into the hardware (and/ or language) environment underlying its particular caplet type. For example, we employ one loader for the XScale caplet and another for the microengine caplets. Importantly, the OpenCOM API allows selective transparency in the use of loaders. If full loader-selection transparency is desired, one can issue a call of the form *load(component_c1, caplet_1)* which will deduce an appropriate loader type from meta-data attached to *component_c1*, and use this to load the component into the designated caplet. This masks the fact that different components may be implemented in different machine languages. Further transparency can be achieved by issuing a call of the form *load(component_c1)* which causes the runtime to load *component_c1* into a default capsule using a default loader (again, the default is selected on the basis of meta-data). Alternatively, one can opt for complete control and zero transparency by issuing a call of the form *load(component_c1, caplet_1, loader _3).*

The pluggable binder concept is equally central to the component model's abstraction power—in this case over the available communication hardware mechanisms available on the IXP2400. For example, we provide a cross-microengine-caplet binder based on the IXP2400's next-neighbour register mechanism. Again, the use of plug-in binders is selectively transparent. If we don't know or care in which caplets our two target components are located, we can say *bind(interface_1, receptacle _15)* and an appropriate loader will be selected according to location-related meta-data attached to the components that own the specified interface and receptacle. On the other hand, if it is important to select a particular mechanism, we can say *bind(interface_1, receptacle_15, loader_4).* And so on.

A final crucial property of the component model is its radically third-party nature in terms of loading and binding. Thanks to this, a component on a microengine can load and bind two components on the XScale, and a component on the XScale can load and bind microengine components using exactly the same syntax as if it were dealing with local XScale components.

As an example of this, and of OpenCOM's ability to abstract over heterogeneity of the IXP2400, consider the following pseudocode segment:

```
template mtemp, xtemp;
comp_inst mcaplet, mloader, xcaplet, xloader,
      mcomp, xcomp, binding, cbinder;
ipnt_inst iface, recpt;

/* load and instantiate the components */
xtemp = load(XSCALECOMP1, xloader, xcaplet);
mtemp = load(MICROCOMP1, mloader, mcaplet);
xcomp = instantiate(xtemplate);
mcomp = instantiate(mtemplate);
```

```
/* bind the two components using
 * a cross-caplet binder
 */
binding = bind(xcomp.iface, mcomp.recpt, cbinder);
```

This example assumes that two caplets have been established: *xcaplet* is a caplet on the XScale and *mcaplet* is a caplet on one of the microengines. The code loads and instantiates two components, one in each caplet, and then binds the two using a cross-caplet binder. Of course, programming would normally be done at the level of component frameworks (see below for an example of this) which raises the level of abstraction still further; but this simple example shows the abstraction power of OpenCOM even at its lowest level. Note especially the 'third-party' nature of the programming model: this code could be executed unchanged with the same effect from within any component in any currently installed caplet. (To execute such code on a microengine caplet a thin proxy transparently forwards the calls and their arguments to the XScale caplet where they are executed under the central capsule runtime.) This means that component developers do not need to know or care which caplet their component will execute in. Furthermore, the programming model is the same regardless of whether the components involved are implemented as Linux shared objects or as blocks of microengine code. Finally, the binding between the two components can exploit the most efficient mechanisms available in the deployment environment (see below): in this case through registers shared between the XScale and the microengines.

4.4. *Implementation of Loaders and Binders*

The mapping we currently employ of OpenCOM capsules and caplets to the IXP2400 involves a single capsule that encompasses both the XScale and the microengines, and contains separate caplets for: i) the XScale (implemented as a single Linux process); and ii) each of the eight microengines. The OpenCOM runtime runs in the XScale caplet; all the other caplets are 'slaves' of this 'central' runtime and incur only minimal memory overheads. The memory footprint of the central runtime itself is of the order of 300KB.

The central runtime in the XScale caplet communicates with the other caplets by means of so-called *caplet channels*. The role of these is to bootstrap plug-in loaders and binders associated with non-central caplets, and to support communication between their two parts: such loaders/ binders are implemented as a 'delegator' part that resides in the central XScale caplet, and a (minimal) 'delegate' part that resides in the other caplet. As examples, we now briefly describe example loader and binder plug-ins that are associated with the microengine caplets.

The *microengine loader plug-in* is of interest in that it provides the illusion of dynamic loading despite the fact that the microengine hardware only allows modification of its instruction store when the CPU is stopped [4]. The basic capability

provided by the microengine hardware is to i) stop the microengine, ii) read/ write arbitrary instruction store locations, and then iii) restart it at a hard-wired address. To achieve transparent dynamic loading it is therefore necessary for the loader to not only load the new component but also to patch the (hard-wired) restart address so that subsequent execution resumes at the point it left off. The loader also has the ability to autonomously move code around within the instruction store to avoid memory fragmentation as components are loaded and unloaded. The loader is also of interest in that it constrains the form of OpenCOM components it is willing to load. The general notion of particular loaders somehow restricting the components they can work with is a powerful feature of OpenCOM. In the present case, the interfaces of loaded components are restricted to supporting operations that accept and return a single integer. This restriction, which is enforced by inspecting the component's IDL meta-data at load time, is imposed partly to simplify the design of the associated binder (see below), and partly because the assumed model of component composition on the microengines is that components are bound into a more-or-less linear sequence and cooperatively work on queues of network packets whose addresses are passed as integer arguments.

Our *intra-microengine binder plug-in* is strongly coupled to the loader just described. It builds on the NetBind-pioneered technique of creating bindings by 'morphing' jump instructions [2]. Together with the loader discussed above, the binder supports multiple instantiations of components (NetBind only supports singleton components). The single argument and return value are passed via a designated register. The necessary entry and exit point information is obtained from IDL metadata attached to the packaged component, which is transformed from relative offsets to absolute offsets by the loader. The overhead of a binding created by this binder in calling a null operation with no arguments or return values is only five (1-cycle) instructions. These subsume passing on the stack a pointer to the per-instance state vector of the called component, and the return address. Note that the performance of the OpenCOM programming model as a whole is almost entirely dependent on the performance of the binding mechanisms used. Almost all the value-added features of OpenCOM are confined to the central runtime and do not 'get in the way' when components communicate with each other on the NP's fast path.

Apart from the microengine loader and binder discussed above, we have a loader that loads components into the XScale caplet; and binders that bind components between and within the two caplet types. Bindings between the microengines and the XScale are more complex than intra- and inter-microengine bindings as they require stubs and skeletons to map the parameter and return value to a bus packet. To minimise memory overhead, the microengine-side stubs/skeletons can be hand coded rather than generated automatically from the IDL specification.

4.5. *Router CF*

We have designed a "Router CF" which accepts, as plug-ins, OpenCOM components that perform arbitrary, user-defined packet-forwarding functions (we also provide "standard" components that interface to network cards and wrap efficient kernel-user space communications mechanisms). All components (see figure 3) are required to conform to the following rules, which are checked by the CF at run-time when the component is loaded:

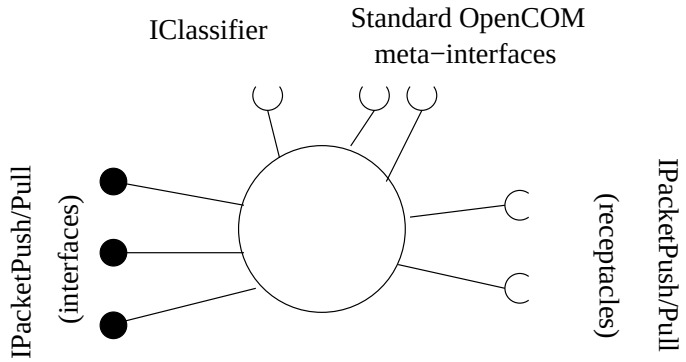

Fig. 3. A valid Router CF component

- compliant components must support appropriate numbers and combinations of specific packet passing interfaces/ receptacles (called *IPacketPush* and *IPacketPull*): these respectively enable push- and pull- oriented inter-component communication); it is possible to dynamically add/ remove instances of these interfaces as long as the CF's rules remain satisfied
- compliant components may (optionally) support an *IClassifier* interface which exports an operation *register_filter()* that is used to install packet-filters; if *IClassifier* is supported, the component must honour the semantics of installed filter specifications in terms of the particular named outgoing *IPacketPush* and *IPacketPull* interface(s) on which each incoming packet should be emitted;
- compliant components may be *composite*, in which case all their internal constituents must (recursively) conform to the CF's rules; additionally composite components should contain a so-called *controller* component that manages the other internal constituents.

The scenarios discussed in the following section use the "Router CF" as a basis.

5. Runtime Reconfiguration on the Intel IXP2400

In this section we illustrate how runtime reconfiguration can be achieved using OpenCOM on the IXP2400. This is illustrated, in 5.1, by revisiting the three scenarios

introduced in section 2. We discuss some general issues arising from consideration of the scenarios in section 5.2.

5.1. *Realising the Scenarios*

5.1.1. *Dynamic Proxying*

We have implemented a dynamic in-band transcoding scenario that is straightforwardly built on top of the programmable classifier discussed above. In-band transcoding (e.g. MPEG) has not been possible in the past without intercepting the multimedia traffic and offloading it to a separate server. As noted earlier, this requires introducing new hardware into the system and also potentially increases the end-to-end latency of media transmission.

In our implementation, a manager component residing in the XScale caplet, programs the classifier by dynamically adding classification rules to detect flows of interest (i.e. as designated by out-of-band control messages from an application). On learning of these, the manager component loads and instantiates a suitable transcoder on the XScale (or a microengine as appropriate). On doing so, the router CF selects an appropriate loader and checks the loaded component (using the interface meta-model) for conformance to its rules. The manager then instructs the classifier to bind to this component and to forward the flow to it. Finally, it uses the architecture meta-model to locate the forwarder and bind the transcoder to it using an appropriate binder. The manager may also choose to use the interception meta-model to add an interceptor to the binding that monitors usage of the transcoder as an input to a policy that determines when to discard the transcoder (e.g. based on resource constraints on the NP and on the amount of traffic in the flow).

This implementation is illustrated in Figure 4 which shows the state of a router with the transcoder manager deployed. The figure shows the reconfigured area (within the dashed line) containing the manager on the XScale and a number of transcoders on both the XScale and the microengines. Note that the microengines can only support primitive transcoders such as frame-droppers; note further, though, that the programming model makes it as straightforward to deploy a transcoder on a microengine as on the XScale.

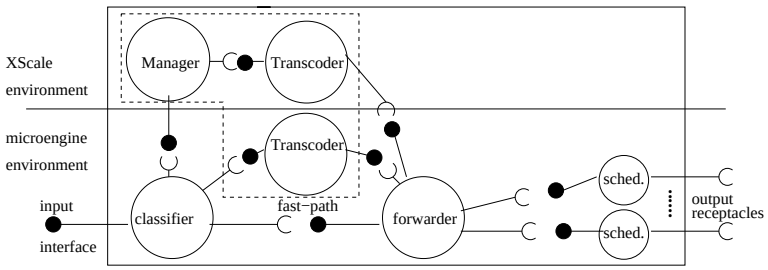

Fig. 4. Transcoding Service mapping to Intel IXP2400

5.1.2. *Adaptive Load Balancing*

The options for load balancing on the Intel IXP2400 are numerous. In typical operation, the bulk of packets traversing the IXP2400 are processed and forwarded by the microengines. At different times in the lifecycle of a typical deployment the load on particular microengines will be increased or decreased. Therefore, to balance increased packet load in the IXP2400 one of the options we have is to replicate packet processing code on additional microengines.

Figure 5 shows the placement of components after a simple method of load balancing has been performed. Before the load-balancing is performed, the top four components constitute the deployment in the microengine. Load-balancing is performed by replicating the "IPv4 header processing" and "forwarding" components on additional microengines. As can be seen from the diagram, the classifier is load balancing across the two chains of components. The dashed line indicates the reconfigured area. This style of load-balancing would be appropriate when there is a significant increase of a type of packet flow which needs additional processing by the NP.

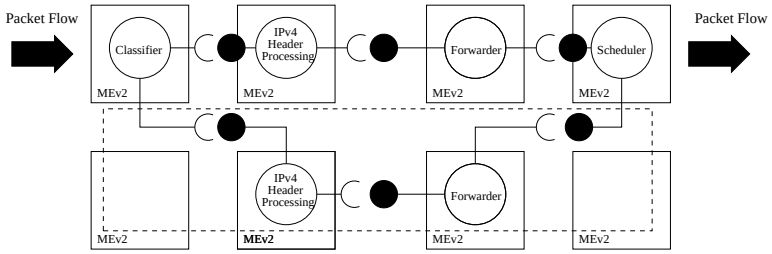

Fig. 5. Simple Load Balancing using Packet Processors

Before processing incoming packets, the classifier diverts a proportion of packet flows to a secondary classifier on a additional microengine. The mechanism of installation of the additional classifier could be either pro-active by a system administrator or reactive by the classifier detecting the increased packet load (either by the primary classifier or by a additional component or system).

This scenario shows that, based on the network situation, the NP control processor can add and remove components to alter the processing capacity of the NP. NPs generically contain a number of packet processors, which perform the majority of in-band packet processing [15]. Therefore this style of load balancing should be applicable in other NP architectures.

In addition to load balancing being performed by the packet processors, it is also possible to load balance with other devices on the internal bus of the IXP2400. This could include the XScale CPU and hardware assist units, but it could also be other IXP2400 NPUs or other NPs. This yields the possibility of more complex load balancing styles than the one illustrated here. The microengines on the IXP2400 also

have the feature of multiple hardware threads, which introduces further possibilities for load balancing within a microengine.

5.1.3. *Network Support for fast Mobile Handoff*

This scenario can be considered as a subset of the dynamic proxying scenario. A difference between this scenario and the previous scenario is that the mobile node itself drives the installation of the flow routing component. For the NP to support fast handoffs it must have logic that detects mobile nodes performing a handoff. The logic required to do this involves examining the mobile nodes packets in greater detail than a classifier would generally do, therefore a additional semi-permanent component is necessary.

The classifier would divert all packets from mobile nodes to the component that detects mobile handoffs. Upon detection of a handoff, a flow routing component is installed onto a unused microengine. Rules will be added to the classifier to continue to route traffic destined for to the mobile client to the flow routing component. The flow routing component will additionally be bound to the classifier and components downstream in the NP. Once the mobile handoff is completed, any traffic mis-routed to the mobile clients old location will be forwarded by the flow routing component to the mobile nodes new location.

The flow routing component would continue to route mis-routed packets to the mobile nodes new address until the mobile handoff has been deemed to be completed and no more packets will be mis-routed. The lifespan of the flow routing component is dependent on the continuation of packets destined for the mobile device. This would be determined by internal policies of the flow routing component. The danger of lost data if the flow routing component is removed would mean that the flow routing component would have to be resident for as long as possible. Therefore the lifespan of the flow routing component will depend heavily on the working capacity of the NPs microengines. This inevitably limited capacity on a NP means that flow routing component may handle more than one fast mobile handoff and thus have a longer lifespan.

5.2. *Performance*

One of the main determining factors in the acceptance of support for runtime reconfiguration on NPs is the overhead incurred. This breaks down into two aspects: i) the overhead of actually performing reconfiguration operations; and ii) the inherent overhead of potentially-reconfigurable software. We discuss these in turn.

The major determinant of the overhead of reconfiguration operations on the IXP2400 is that the microengines need to be stopped before new code can be loaded. Our measurements show that to halt, update and restart a microengine takes a total time of 60ms. Our IXP2400 development board contains 3 OC-48 ports which can each deliver 2.488Gbps, a total of 7.464Gbps. A delay in the system would therefore require a maximum theoretical of 56MB to buffer all incoming packets and avoid

dropping packets, well within the means of a NP. Furthermore, wholesale reconfigurations of all the microengines would be relatively uncommon. More likely would be a localised reconfiguration of a single microengine or a group of microengines. In this case it would be necessary only to buffer the packets being passed to the component(s), this would be considerably less than that of reconfiguring the whole system. It may not even be necessary to perform any buffering, especially in scenarios where reconfiguration is infrequent and where TCP can be relied on to recover from network-level packet loss.

Furthermore, wholesale reconfigurations of all the microengines, or of the classifier, requiring a complete system halt would be relatively uncommon. More likely would be a localised reconfiguration of a single microengine or a group of microengines, like those illustrated by the scenarios in the previous sections. In this case it would be necessary only to buffer the packets being passed to the component(s), this would be considerably less than that of reconfiguring the whole system. It may not even be necessary to perform any buffering, especially in scenarios where reconfiguration is infrequent and where TCP can be relied on to recover from network-level packet loss.

The second factor to be considered is the inherent overhead of potentially-reconfigurable component-based software; mainly attributed to the bindings between components. To evaluate the throughput overhead of instantiating OpenCOM components on the IXP2400, we utilised two Dell Precision 340 Workstations with 2Ghz P4's and 512MB RDRAM running Linux 2.6.12. Two 3COM 3C996-SX network interface cards were used to send and receive packets through a Radysis ENP-2611 which consists of a IXP2400 NP and 3X 2.5Gbps fibre ports. The XScale CPU of the IXP2400 was bootstrapped with Linux 2.6.11 and all microengine code was loaded and bound from an OpenCOM instantiation on the XScale CPU. The following throughput results where collected using the Thrulay tool which uses a client/server approach to measure TCP throughput.

A single OpenCOM component which performed a simple layer 2 bridging operation between two fiber channel connections was deployed on a single thread of a microengine. The component was capable of processing packets at a sustained rate of 632.42Mbps end-to-end compared to a monolythic Intel implementation at 632.81Mbps (also running on a single thread of a single microengine). Additional 'null' components were then instantiated directly in the data-path between the send and receive portions of the bridging component. To isolate the effect of these components on the throughput of the bridging operation, the components were instantiated in the same thread and microengine of the bridging component.

Table 1 presents end-to-end throughput and latency figures for different numbers of 'null' OpenCOM components instantiated on the IXP2400 as described. It shows that the overhead of inserting five or less OpenCOM microengine components is minimal, inserting between 10 and 20 components introduces a sizable lag into the system, this might be considered acceptable for the advantages offered. The insertion

Table 1. Throughput and Latency of OpenCOM Microengine Components

Number of Components	Throughput	Latency
Intel Implementation	632.81Mbps	0.52ms
1 Component	632.42Mbps	0.54ms
5 Components	614.86Mbps	0.59ms
10 Components	603.25Mbps	0.64ms
15 Components	589.82Mbps	0.88ms
20 Components	567.39Mbps	1.17ms
50 Components	496.03Mbps	2.81ms
100 Components	435.72Mbps	3.65ms

of ten or more components introduces a sizable lag into the system which would be considered unacceptable for a high-speed router. In addition the figures for latency correlate with the throughput figures with increasing latency with more components.

However, in a real world deployment there would be many different flows requiring processing by different sets of microengine and XScale components, thus the effect of adding a single component would be reduced. The implication of these results is that the most effective way to deploy OpenCOM components is using multiple short pipelines of five or less components.

6. Related work

NetBind [2] provides the abstraction of a set of packet-processing components that can be bound into a data path. This is done by adopting the convention of a standard entry and exit instruction sequence for microblocks, and offering the capability to dynamically 'morph' jump instructions in these sequences so that execution is transferred to the entry point of the microblock to be executed next. This separates the raw functionality of a microblock from the way it is composed with others, and also gives the NetBind programmer the ability to dynamically reconfigure compositions of microblocks.

NetBind goes beyond MicroACE in supporting flexible composition of microblocks, but it offers no abstraction over the NP's memory organisation, interconnects, or over different sorts of processors (e.g. the microengines, XScale, and workstation host of an IXP1200-based router). It therefore offers no more design portability across different NPs than MicroACE.

NP-Click [18] is another component-based programming model for NPs; it is derived from an earlier PC-based software router model called *Click*. Again, NP-Click has been primarily targeted at the IXP1200. It is founded on a much richer model of components than NetBind. While communication between NetBind microblocks takes place over low-level untyped entry and exit points, Click components have typed *ports*; and connections between ports can be designated as either 'push' or 'pull' which provides declarative control over flow of control and threading. In addition, NP-Click abstracts, to a degree, over the different memory technologies offered

by the IXP1200 by providing keywords such as 'global', 'regional' or 'local' which cause the associated component to be automatically allocated an appropriate memory type. Furthermore, it provides low level abstractions such as *malloc()* and *free()* to facilitate and manage the allocation of NP resources such as microengine LIFO registers.

NP-Click does a much better job of abstracting NP architecture than NetBind, but it still falls short of providing a generic approach to NP programming. While it abstracts particular features of the IXP1200, it has no notion of abstracting arbitrary architectures in a principled way, and thereby encouraging design portability and transferable skills across NP types. That is, there is no necessary commonality between the abstractions provided over different architectures (e.g. NPs other than the IXP1200 may not use LIFOs). In addition, NP-Click provides no support for dynamic reconfiguration.

VERA [19] is an extensible software router architecture that comprises three layers: the top layer provides the abstraction of a router, the bottom layer abstracts the hardware, and a middle 'distributed operating system' layer mediates between the two. The latter layer organises the available packet processors into a hierarchy of levels. Initial classification occurs on a 'low level' processor attached to the MAC-port, and if the packet requires further or more complex processing then a 'higher level' processor is used. This provides a high degree of abstraction, but it is heavily dependent on the IXP1200 architecture.

7. Conclusions

We have argued that developing NP software with support for runtime reconfiguration enables the full potential of NPs to be realised, and that this yields significant benefits for high-speed routing platforms. More specifically, we have introduced a number of runtime reconfiguration scenarios for NP platforms and showed how they can be implemented on the Intel IXP2400 using our OpenCOM programming platform.

We also argue the approach we outline is in principle applicable not only to the IXP2400, but to a range of NP architectures. This claim is made on the basis of the generality of the OpenCOM platform as discussed in 4.3 and on the basis of a study of the mapping of OpenCOM to other NP architectures [15]. Future work will include the exploration of using OpenCOM in other NP environments to provide further evidence for the generality of our approach.

References

1. Y. Bernet and J. Binder et al. A framework for differentiated services. In *draft-ietf-diffserv-framework-02*, 1999.
2. A. Campbell, M. Kounavis, and D. Villela. NetBind: A Binding Tool for Constructing Data Paths in Network Processor-based Routers. In *IEEE International Conference on Open Architectures*, pages 91–103, June 2002.

3. D. Comer. Network Systems Design using Network Processors, IXP edition. 2003.
4. Intel Corporation. MicroACE, Design Document, revision 1.0. *Intel Corporation*, 2001.
5. Intel Corporation. Introduction to the Auto-Partitioning Programming Model: Accelerating custom dpplication development for the Intel IXP2XXX network processors. *Intel Press, Intel Corporation*, 2003.
6. Intel Corporation. Intel IXP2400 Network Processor. In *Datasheet 301164-011*. Intel Corporation, Feb 2004.
7. G. Coulson, G. Blair, P. Grace, A. Joolia, K. Lee, and J. Ueyama. A component model for building systems software. In *IASTED 2004*, Cambridge, MA, USA, 2004.
8. A. Deshpande, K. Crozier, and M. Baines. The Teja Software Platform for Network Processors, Teja Technologies. In *www.teja.com*, 2001.
9. C. Lee et al. Software/hardware reconfigurable network processor for space networks. In *4th annual Military and Aerospace Applications of Programmable Devices and Technologies International Conference (MAPLS)*, Jan 2001.
10. A. Gavrilovska, K. Schwan, and O. Nordstrom. Network processors as building blocks in overlay networks. In *11th Symposium on High Performance Interconnects*, pages 83–88, Aug 2003.
11. J. Hagino and K. Yamamoto. An IPv6-to-IPv4 Transport Relay Translator. RFC 3142 (Informational), June 2001.
12. S. Harper. Phd thesis: A Secure Adaptive Network Processor, Bradley Department of Electrical and Computer Engineering Blacksburg, Virginia, April 2003.
13. Heavy Reading. Network processors: A heavy reading competitive analysis. In *Vol. 3, No. 2*, January 2005.
14. A. Kind, R.Pletka, and M.Waldvogel. The role of network processors in active networks. In *International Working Conference on Active and Programmable Networks (IWAN)*, pages 18–29, Japan, 2003.
15. K. Lee, G.Coulson, and G. Blair et al. Towards a generic programming model for network processors. In *IEEE International Conference on Networks*, pages 504–510, Singapore, Nov 2004.
16. L. Ruf, R. Keller, and B. Plattner. A Scalable High-performance Router Platform Supporting Dynamic Service Extensibility On Network and Host Processors. In *IEEE Conference on Pervasive Services*, pages 199–206, Jul 2004.
17. S. Schmid, J. Finney, A. Scott, and D. Shepherd. Active component driven network handoff for mobile multimedia systems. In *Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services*, pages 266–278. Springer-Verlag, 2000.
18. N. Shah, W. Plishker, and K. Keutzer. NP-Click: A Programming Model for the Intel IXP1200. In *2nd Workshop on Network Processors at HPCA-9*, pages 100–111, Anaheim, February 2003.
19. T. Spalink, S. Karlin, L. Peterson, and Y. Gottlieb. Building a Robust Software-Based Router Using Network Processors. In *In Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, pages 216–229, Oct 2001.
20. I.A. Troxel, A.D. George, and S. Oral. Design and analysis of a dynamically reconfigurable network processor. In *IEEE Conference on Local Computer Networks*, pages 483–494, Nov 2002.